(NeXT Tip #18) Don't avoid 'void'

Christopher Lane (*lane[at]CAMIS.Stanford.EDU*)
Mon, 1 Feb 1993 15:03:57 -0800 (PST)

To: KSL-NeXT[at]CAMIS.Stanford.EDU
cc: KSL-HP[at]CAMIS.Stanford.EDU
Message-ID: <MailManager.728607837.2191.lane[at]ssrg-next-1>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII

(This is a generic ANSI-C style tip. Like any code style issue, there is no one 'right' way to do things and several dozen wrong ways. :-)

The 'void' keyword in C programming serves several purposes where it's conceptual meaning ranges from 'nothing' to 'anything'. I'll try to describe some of it purposes and how/why/when you should use it:

1) The 'void' datatype

The 'void' datatype signifies that a function returns nothing or has no arguments. The best known example is the obligatory 'main()' function:

```
void main(int argc, char **argv);
```

Since 'main()' calls 'exit()' to return directly to the calling process, it's declared not to return anything. (It it also historically legal to declare 'main()' as returning an 'int' and then have it return a value rather than 'exit()', but these two schemes shouldn't be mixed -- use one or the other.)

Do not use 'void' to declare that Objective-C methods don't return anything, such methods should always be of type 'id' and 'return self'. Here's an example of such from NeXT's NXColorList.h:

```
- (void) removeColorNamed:(const char *) colorName; // DON'T DO THIS!
```

This prevents you from chaining method invocations. One exception to do this is if the method destroys the object on which it is invoked, and thus chaining methods would be a serious programming error -- declaring it 'void' will let the compiler warn you. Another exception is building an asynchronous method for a remote distributed object, since it will never return.

The 'void' datatype can be used to declare that a routine takes no arguments:

```
extern int getchar(void);
```

Under ANSI C, this is different from declaring the routine as:

```
extern int getchar();
```

which says that 'getchar()' returns an 'int' and takes some unknown number of arguments. (Of course, you do not use 'void' to indicate an Objective-C method takes no argument as the Objective-C syntax itself does that.)

2) The 'void' cast

Another use of the 'void' datatype is to nullify return values, for example:

```
(void) strcpy(buffer, argv[FILENAME]);
```

The 'strcpy()' routine normally returns a pointer, i.e. it's first argument, but I'm telling the compiler, and anyone else reading my program, that, "Yes, I know it returns something -- I chose not to use it." This is important, style-wise, for routines that return error codes that you don't check:

```
(void) vm_allocate(task_self(), (vm_address_t *) &memory, size, TRUE);
```

The 'vm_allocate()' routine returns an error code on failure and by flagging that I've chosen not to look at it, I've left an indication where the program might need to be examined when problems occur. However, do not cast to 'void' functions that are already declared to return 'void', like 'NXClose()'.

Although casting to 'void' also applies to Objective-C methods, don't cast to 'void' methods that return 'self', the default, only those that return something other than 'self' which you don't choose to do anything with:

```
(void) [pasteboard types]; // initializes PasteBoard as a side effect
```

One possible exception to this style rule is 'printf()' and its variations. These functions are defined to return 'int' but what that integer means isn't defined. So, since these routines are called often in some programs, the choice to '(void)' the result is up to you. (Be consistent.) Personally, I '(void)' it in protest of it not being defined it to return something useful.

3) The 'void *' datatype

A pointer to a 'void' type isn't a pointer to nothing -- it's a pointer to anything. This is important for routines that take a pointer to something unknown, either along with a function that casts it to a real type or an implicit understanding about how the pointer will be used.

A typical example is 'qsort()' which sorts data. It's first argument is a 'void *' as 'qsort()' itself never dereferences that pointer -- it hands it to a comparison function, one of its other arguments, that knows what the real type of the initial pointer argument is. Another is:

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream)
```

which copies data to wherever 'ptr' references, regardless of its type, since information normally derived from the type, i.e. the size of what it points to, is an explicit argument to the function.

A 'void *' pointer is defined to be large enough a datatype to hold, when cast to '(void *)', any other pointer type. So any pointer datatype can be cast to '(void *)' and vice versa. This is why 'malloc()' is defined:

```
extern void *malloc(size_t size);
```

and you usually cast its result to the actual pointer type you need. When releasing the memory via 'free()', you cast it back to a '(void *)' pointer.

Although a '(void *)' pointer can hold any other pointer type, it cannot hold just any other non-pointer type. You'll see code that stores non-pointers in '(void *)' datatypes, like 'int', 'short', 'float', etc. because they happen to fit on the architecture in question, but they're not guaranteed to fit. (E.g. on the NeXT, you can't store 'double' types in a 'void *' as they're too large.) Any code that casts non-pointers to '(void *)' is suspect when moving to a different hardware platform or compiler. (See the source for my Message.app for an example where non-pointer coercion to 'void' is necessary.)

Never dereference a 'void' pointer, always cast it to a real pointer first:

```
int compare(void *a, void *b) { return *((char *) a) - *((char *) b)); }
```

This silly routine takes two string pointers, passed in as 'void' pointers and returns < 0, 0, or > 0 based on just comparing just initial characters and is suitable for passing to 'qsort()'.

4) Notes

The 'void' type was not originally in C and started getting used before it became part of the standardized ANSI C. In code for non-ANSI compilers, you'll sometimes find it defined as a synonym for 'char *' as it was previously defined as being large enough to hold any other pointer type. That's why non-ANSI compilers define 'malloc()' to return a 'char *'.

Why is having a pointer type large enough to hold other pointer types significant? On some machines, all pointer types aren't the same size, for example a 'char *' might be a full machine word whereas a pointer to a word aligned datatype might only require half of a machine word. Since you can't know what the size will be (and shouldn't need to), ANSI C defines the void pointer to be as large as any other pointer type the compiler generates and thus is a safe type to cast other pointer types into and out of.

In GNU C, addition and subtraction operations are legal on void pointers -- avoid doing this as it's compiler specific. (Not to mention confusing.)

- Christopher 'Or cast as rubbish to the void -- Tennyson'